CS454 Assignment 2: Text Ranker
Assigned: Thursday, October 27, 2005
Due: 4:00 PM, Monday, November 7, 2005
(Turn in hardcopy part to Alicen Smith, CSE 546, or place under her door)

1 Project Description

Text Ranker is a program that allows the user to query a set of web documents, just as any search engine does. As a general rule there will be more hits than any user would want to examine, so ranking the hits is critical.

Your job is to write the code that determines the "relevance" of each document to the query in question. The rest of the search engine has been written for you.

2 Project Objectives

By the end of this project you should understand some classical (pre-Web) information retrieval techniques, and gain some insight into what makes a good ranking function. WIth luck, this project shouldn't involve as much programming as PS1, but it will involve considerable experimentation and tuning, which may be time-consuming. Don't wait until the last minute!

You should start by implementing TF-IDF as described in class slides. Find some queries that seem to generate particularly good or bad results.

Try to fix some of the problems you find by writing a new ranking function. Some questions you might want to think about include:

- What is the intuition/motivation behind tf-idf? Does your function still capture it?
- The web contains all sorts of strange documents your ranker needs to handle. For example, consider a single-word document that contains only the word "antidisestablishmentarianism." What will tf-idf do with this document? Will the search user find this useful? Can you improve things?
- Also consider a "dictionary" document that consists of every term in the English language repeated, say, 100 times each. What will tf-idf do? What is a better approach?
- The CSE454 engine only checks the document body for hits, not the document URL. How can you use the URL to improve ranking? What might be a reasonable tokenization of the URL?
- The tf-idf function wholly ignores term positions within the document. Is this what users expect? Can you improve on this?

You can probably think of other reasonable ideas a ranking function might contain. Good rankers inevitably have many parts, some of which can be in conflict. For example, as hinted above, extremely short documents are often not useful even though they can contain extremely rare terms.

Be sure your program is well-documented. Each method and class should have a header comment describing what's going on. The ranking formula, especially, should have *very* detailed comments. An independent reader should be able to understand what your code is trying to do and why.

After you have settled on a ranking function that seems to work best, write a short paper about your decisions. Describe each part of your function and your motivations for including it. Make sure your code, your code comments, and your paper are consistent; they will be read and considered together.

Your paper should also include some brief experimental results. Include a few queries (with results) that illustrate why certain aspects of your ranker are desirable. "Before-and-after" document rankings, generated by the same query, would be particularly helpful in justifying your decisions.

3 Getting Started

Before we can write the ranker, we need to figure out how to run the rest of the engine.

3.1 The CSE454 Search Engine

You can invoke a command-line version of the CSE454 Search Engine with the following command: `/projects/instr/cse454-05au/assignment2/bin/cse454`

Try running this program, and you'll see this:

```
Usage: cse454 COMMAND
where COMMAND is one of:
  query             Query the CSE454 crawl, using your ranking code.
Commands print help when invoked w/o parameters.
More commands will be available in future assignments.
```

It's a little silly to have only one option here, but there will be more later on in the course. So now run the program with the parameter "query" and you will see:

```
Usage:  cse454 query [-maxHits n] -query queryTerm0 (queryTerm1 ....)  [-rankerArgs
rankerArg0 (rankerArg1 ....)]
```

This command will issue a single query, computed over one of two standard class document sets. (Right now the small set consists of just under 90,000 docs, and the large set will be announced later.)

Unfortunately, the engine can't find its ranker right away. You need to tell it how to find one. After you compile the class `edu.washington.cse454.DocumentRanker`, you can tell the engine about it by setting the CSE454_CLASSPATH environment variable. Set this value to the directory that contains the `DocumentRanker` class, and you'll be ready to go. [1]

[1] Well, that's almost true. Remember that Java places classes in directories that are generated from the class package name. So Java expects that the `edu.washington.cse454.DocumentRanker` class

Let's assume for a minute we have a very bad ranker implementation in place. We can run a query with the above command and see output like this:

```
Document set has 45 docs and 7295 unique terms.
Searching for terms 'yahoo games' ...  found 20 hits (displaying top 10).
1.  http://www.yahoo.com/index.html (0.0)
2.  http://www.yahoo.com/s/99361 (0.0)
3.  http://www.yahoo.com/r/en (0.0)
4.  http://www.yahoo.com/s/97664 (0.0)
5.  http://www.yahoo.com/s/105306 (0.0)
6.  http://www.yahoo.com/r/sp (0.0)
7.  http://www.yahoo.com/s/105305 (0.0)
8.  http://www.yahoo.com/s/143682 (0.0)
9.  http://www.yahoo.com/s/97665 (0.0)
10.  http://www.yahoo.com/r/mu (0.0)
```

The first line indicates the size of the document set and is the same for all queries. (Note these results are not from the crawl you will be using.)

The second line indicates the list of query terms, plus how many hits the searcher found. A query can consist of an arbitrary number of terms. Like most search engines these days, a multi-term query will only hit documents where ALL terms are present. The CSE454 engine does not currently offer quoted-string queries or fancy Boolean formulae. Queries are case-insensitive.

The ranked list is where the action is, and where your code comes in. The URLs listed are all those for which we found a hit. They are sorted in descending order by the "relevance score," found in parentheses. It's hard to do worse than the ranker seen here - it's assigning zero to every hit. (This is the default behavior of the search engine when you get to it.) The engine will present documents with tied relevance in arbitrary order.

The next section covers how your code can improve on this default ranker.

3.2 Interfaces

You can find skeleton code for `edu.washington.cse454.DocumentRanker` by looking here:
`/projects/instr/cse454-05au/assignment2/samples/DocumentRanker.java`

This file is reproduced in Appendix A.

You should copy this file to your work area and use it as a starting point.

Take a look at all the relevant interfaces here:
`/projects/instr/cse454-05au/assignment2/reference`

---

will appear in a file called `DocumentRanker.class` in a directory hierarchy named `edu/washington/cse454/` . If your compiler is outputting to a directory called `classes`, and you have your compiled class at `classes/edu/washington/cse545/DocumentRanker.class`, the CSE454_CLASSPATH variable should be set to `classes`. The Java runtime will look in the right subdirectory.

3.2.1 IRanker

The `DocumentRanker.java` class implements the following interface:

```
public interface IRanker {
    /** Called once, upon object creation.  **/
    public void init(DocumentSetInfo info);

    /** Called every time the search engine finds a 'hit' **/
    public double getRelevance(String queryTerms[], Document doc);
}
```

When the CSE454 search engine starts up, it will call the IRanker.init() just once. The passed-in `DocumentSetInfo` object is pretty useful, so you'll probably want to store it.

The search engine will call `getRelevance` for every hit it finds. If the query hits no documents at all, the `getRelevance()` function will not be called.

3.2.2 DocumentSetInfo

Here is the interface for the above-mentioned `DocumentSetInfo`, passed in at initialization:

```
public interface DocumentSetInfo {
    /**
     * Returns an array of Strings passed in on the command-line.
     * This lets you pass arguments to your DocumentRanker, which might
     * be useful for running tests.
     *
     * Of course, your final code should work correctly *without* any
     * args being passed-in.
     *
     * If no args were given on the command line, this function returns
     * a zero-length array.
     */
    public String[] getRankerArgs();

    /**
     * Returns how many docs there are in the overall index.
     */
    public int numDocs();

    /**
     * Find how many indexed docs there are containing the given term.
     */
    public int numDocsContainingTerm(String term);
```

```
    /**
     * Find how many times the term appears in the document set (possibly
     * multiple times per page).
     */
    public int numTermOccurrences(String term);
}
```

3.2.3 Document

There's also the `Document` class, which is passed in whenever the engine wants you to rate a doc's relevance:

```
public interface Document {
    /**
     * Simply return the number of terms that appear in the Document.
     * Note that for efficiency's sake this number might not be completely
     * accurate, but it will be very close.
     */
    public int getDocLength();

    /**
     * Returns how many times the given term occurs in the Document.
     * Must be one of the original query terms.
     */
    public int freq(String term);

    /**
     * Returns an array of every position the given term occurs in the
     * document.  This information can be used to compute proximity between
     * query terms.
     *
     * If the term was not one of the original query terms, returns null.
     */
    public int[] positionsInText(String term);

    /**
     * This is the normalizer of the tf-idf equation, computed with a base-10 log.
     */
    public float getNormalizer();

    ////////////////////////////////////////////////////////////////////////
    // The following methods are very useful, but are extremely
    // time-consuming to call.  You can recreate most, but not all, of their
    // functionality using the methods above.  There is little reason
    // now to call getText().
    //
```

```
    // If you really want to call one of the below methods, you should
    // avoid doing so inside every getRelevance() call.  For example,
    // you might only examine the URL if the doc's score is otherwise
    // unusually low or high.
    //
    ////////////////////////////////////////////////////////////////////

    /**
     * Get the URL of the current Document
     */
    public String getURL();

    /**
     * Get the extracted title text of the current Document.
     */
    public String getTitle();

    /**
     * Get the full un-tokenized content of the Document.
     */
    public String getText();
}
```

Note that it's not at all necessary to use every method in order to get excellent ranking.

4 What to Hand In

- Your implementation of DocumentRanker, with extensive comments. Be sure your ranking function is clear and easy to understand.
- A description of your ranker, no more than 4 pages in length. Make sure that the names and contact information for both team members are listed on the first page of the report. Describe the various problems your ranker tries to solve, and how each element of the ranking function helps with the solution. Include a comparison of your ranker with the tf-idf function described in class slides.

  Be sure your description of your ranker and your choices behind it match your code and your code comments. Also, include sample queries and results that illustrate your points.
- Turn in electronic copies of these materials and a hardcopy of the report (See the beginning of this handout for hardcopy turn-in instructions. Double-sided printing prefered, if possible).

We will supply details on the handin process via the class mailing list.

5 Grading

The grade for this assignment will be computed as follows:

- Clear and sensible algorithm, cleanly implemented and described, 45 %
- Clear and convincing writeup, 35 %
- Ranking performance on human-judged test set, 20 %

The last element on the list consists of a set of ranking tests, compared against a set of human (well, TA)-judged rankings. There is one set of test queries that will be applied to every project, but this list will not be revealed until after the deadline (so that rankers do not overfit the test set).

Late assignments will be penalized as described in the class policies at http://www.cs.washington.edu/education/courses/454/05au/policies.html.

6 Groups and Collaboration

You will work in pairs on this first project. (Later projects will be collaborative.) Discussions among different students are allowed, subject to the Giligan's Island rule and other important directives listed in the class policies.

7 Appendix A

Here is the skeleton code found at
`/projects/instr/cse454-05au/assignment2/samples/DocumentRanker.java`:

```
package edu.washington.cse454;

import edu.washington.cse454support.*;

/*************************************************************
 * DocumentRanker is an unfinished class you must implement for
 * your search engine to work correctly.  Most of the search
 * engine code already exists; you 'only' need to write the
 * code that ranks documents by "relevance".
 *
 * Whenever the engine needs to find how "relevant" a page is
 * in relation to a query, it will call DocumentRanker.getRelevance().
 * The hit set will then be sorted according to the values that
 * you return from the getRelevance() function.
 *
 * This class *must* implement the IRanker interface, or else
```

```
 * it will not work with the CSE454 framework code.
 *
 * For more info, see the course webpage at
 * http://www.cs.washington.edu/education/courses/454/
 ************************************************************/
public class DocumentRanker implements IRanker {
    /**
     * The constructor for your class.
     *
     * NOTE: It is *CRITICAL* that this constructor have no arguments.
     * Otherwise, the CSE454 framework will not be able to use your code.
     */
    public DocumentRanker() {
    }

    /**
     * The init() method is called immediately after the class
     * is created, and is called only once.
     *
     * It provides your DocumentRanker instance with a DocumentSetInfo
     * object.  This object provides various statistics about the indexed
     * documents and terms.
     */
    public void init(DocumentSetInfo info) {
    }

    /**
     * getRelevance() returns a double.  The higher the value returned,
     * the more relevant the document.
     *
     * This function is called once for every "hit" document found
     * (no matter how many hits are found in a single doc).  If
     * there are no hits for your query, this method will not be
     * called at all.
     *
     * Defining this method usefully is the meat of Assignment 1.
     */
    public double getRelevance(String queryTerms[], Document doc) {
        return 0.0;
    }
}
```